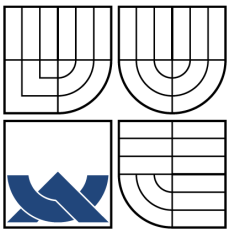
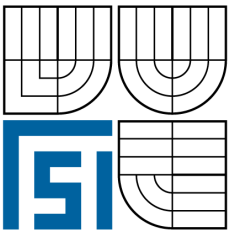


VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ
BRNO UNIVERSITY OF TECHNOLOGY



FAKULTA STROJNÍHO INŽENÝRSTVÍ
ÚSTAV MATEMATIKY
FACULTY OF MECHANICAL ENGINEERING
INSTITUTE OF MATHEMATICS

ANALYSIS OF DATA FLOW IN THE FLY-BY-WIRE SYSTEM

ANALÝZA DATOVÉHO TOKU VE FLY-BY-WIRE SYSTÉMU

DIPLOMOVÁ PRÁCE
MASTER'S THESIS

AUTOR PRÁCE
AUTHOR

Bc. ZUZANA KUBÍNOVÁ

VEDOUCÍ PRÁCE
SUPERVISOR

Ing. RADOMIL MATOUŠEK, Ph.D.

BRNO 2010

Abstrakt
Software implementovaný v rámci Fly-by-Wire systému firmy Honeywell je složen z mnoha modelů implementovaných v systému Simulink. Tato práce se zabývá hledáním přímých a nepřímých spojitostí mezi jednotlivými signály v systému těchto modelů. Jako vhodný aparát je zvolena teorie grafů. Na základě modelů jsou vygenerovány grafy a úloha nalézt spojitost mezi signály je převedena na úlohu nalézt cestu mezi dvěma vrcholy grafu. Známé algoritmy k vyhledávání cest v grafech určují délku nebo váhu nalezené cesty. Pro tuto aplikaci je ale potřeba rozlišovat cesty primární a sekundární bez ohledu na jejich délku. Proto jsou tyto známé algoritmy upraveny.

Summary
The Software implemented within Honeywell Fly-by-Wire system consists of many models implemented in Matlab Simulink. This thesis describes searching for direct or indirect relations among particular signals within the model system. As a convenient apparatus the graph theory was chosen. Graphs are generated according to the models and the problem of searching relations among signals is transformed to a problem of searching paths between vertices of a graph. Common search path algorithms determine a length or a weigth of found paths. For this application it is necessary to distinguish primary and secondary paths not considering their lengths. Therefore the algorithms are modified accordingly.

Klíčová slova
Fly-by-Wire, BFS, DFS, Dijkstrův algoritmus, Floydův-Warshallův algoritmus

Keywords
Fly-by-Wire, BFS, DFS, Dijksta's Algorithm, Floyd-Warshall Algorithm

KUBíNOVÁ, Z.*Analysis of Data Flow in the Fly-by-Wire System.* Brno: Vysoké učení technické v Brně, Fakulta strojního inženýrství, 2010. 46 s. Vedoucí diplomové práce Ing. Radomil Matoušek, Ph.D..

I declare that the thesis *Analysis of Data Flow in the Fly-by-Wire System* is a result of my own work under supervision of Ing. Radomil Matoušek, Ph.D., with a use of materials listed in the Bibliography.

Bc. Zuzana Kubínová

I would like to thank Ing. Radomil Matoušek, Ph.D. for supervision of my thesis, Ing. Michal Závišek for technical consultations of the Fly-by-Wire system and Ing. Roman Hubáček for programming supervision.

Bc. Zuzana Kubínová

Contents

1 Introduction **3**

2 Flight Control **4**
2.1 Flight Control System . . . 4
2.1.1 Mechanical . . . 4
2.1.2 Hydromechanical . . . 4
2.1.3 Fly-by-Wire System . . . 5
2.2 Embedded System . . . 6
2.3 Model Based Design . . . 6
2.4 Honeywell Fly-by-Wire . . . 7

3 Theoretical Background **8**
3.1 Graphs . . . 8
3.2 Networks . . . 10

4 Search Path Algorithms **11**
4.1 Complexity of Algorithms . . . 12
4.2 Algorithms . . . 13
4.3 Shortest Paths Problem . . . 14
4.3.1 Breath First Search Algorithm . . . 15
4.3.2 Algorithm of Dijkstra . . . 17
4.3.3 Algorithm of Floyd and Warshall . . . 18
4.4 Depth First Search Algorithm . . . 20
4.5 Representation of Graphs . . . 21

5 Models **23**
5.1 Model . . . 23
5.2 Model versus Graph . . . 23
5.3 Blocks . . . 23
5.4 Special Case Blocks . . . 25
5.4.1 Inports and Outports . . . 25
5.4.2 Muxes, Demuxes and Logic of Muxed Signals . . . 26
5.4.3 Subsystems . . . 28
5.4.4 Enable block . . . 29
5.4.5 Stateflow . . . 30
5.4.6 Blocks Goto and From . . . 31
5.4.7 Selector Block . . . 31
5.5 Edges . . . 33
5.5.1 Blocks' Inner Connections . . . 33
5.5.2 Blocks' Connections . . . 33
5.5.3 Virtual Connections . . . 34
5.5.4 Weight of Edges in the Graph . . . 35

6 Computer Program **37**
6.1 Programming Languages 37
6.1.1 Graphviz - Graph Visualisation Software 37
6.1.2 MATLAB 37
6.1.3 Perl 37
6.2 Searching Paths 38
6.3 Results 39
6.3.1 Algorithm of Floyd and Warshall 39
6.3.2 Modified BFS, DFS, Algorithm of Dijkstra 40
6.4 Program Manual 41

7 Conclusion **44**

1. Introduction

The Fly-by-Wire system is a new trend in aircraft flight control systems. In the Fly-by-Wire system the mechanical parts of a flight control system are replaced by electronic parts. The electronics is operated either by an analog controller or a digital computer.

For the digital computer a sophisticated software is necessary to be able to deal with complexity of aerodynamics problems. To decrease time necessary for developing and testing the software the model-based design is used. It means that the software is developped as a system of models implemented in some model-based control design software. One of advantages of the model-based design is that engineers can work simultaneously. However, this can also be a disadvantage because one engineer knows only their models. If the engineer needs to know some functionality described in other models, they need to go through all the models to find what they need, and it can be time consuming. For this reason Honeywell has internally developped a software being able to trace a signal through the model system. However, this software does not describe the signal flow inside models. That is a goal of this thesis.

A Simulink model is a block diagram with some added features. This suggests that the graph theory could be used to trace and describe the signal flow inside models. An analysis of Simulink models shows that the models cannot be identified as graphs. Therefore graphs are constructed according to the analysis and the problem of tracing the signals inside models is transformed to a problem of searching paths in graphs.

According to the requirements of Honeywell engineers, it was decided not to search for shortest or longest paths, but for "primary" or "secondary" paths. If a path describes flow of an input signal data, which with some modifications appear in an output signal data, then the path is primary. If a path describes flow of an input signal data, which modify some other data that appear in an output signal, then the path is secondary.

This thesis introduces selected pieces of knowledge of the graph theory necessary for searching paths in graphs and propose modifications of known search path algorithms to be able to distinguish primary and secondary paths. As a part of this thesis also a program creating graphs according to the models is implemented and also a program for searching the primary and secondary paths in the graphs using the modified algorithms is implemented.

2. Flight Control

The primary function of any control system is to convey instructions from the operator to the operated device, in an aircraft, from the pilot to the vehicle. In this chapter some more information about flight control is presented.

2.1. Flight Control System

An aircraft flight control system needs to have capability of controlling the forces and moments on the vehicle. When these are controlled, it is possible to control accelerations and hence velocities, translations and rotations. The forces and moments are affected by control surfaces and by engines, which provide the thrust causing other forces and moments acting on the vehicle. Besides control surfaces and engines, the respective cockpit controls, connecting linkages and the necessary operating mechanisms to control an aircraft's direction in flight are considered as parts of the flight control system. Basic types of flight controls are mechanical, hydromechanical and fly-by-wire system.

2.1.1. Mechanical

Mechanical or manually-operated flight control systems are the most basic method of controlling an aircraft. They were used in early aircraft and are currently used in small aircraft where the aerodynamic forces are not excessive, for example in ultralight airplanes. Very early aircraft used a system of wing warping where no control surfaces were used. A manual flight control system uses a collection of mechanical parts such as rods, tension cables, pulleys and sometimes chains to transmit the forces applied to the cockpit controls directly to the control surfaces. Turnbuckles are often used to adjust control cable tension. Some aircraft have gust locks fitted as part of the control system.

Increase in the control surface area required by large aircraft or higher loads caused by high airspeeds in small aircraft led to a large increase in the forces acting on the surfaces and hence increase in the forces needed to move them, consequently complicated mechanical gearing arrangements were developed to extract maximum mechanical advantage in order to reduce the forces required from the pilots.

2.1.2. Hydromechanical

Larger and heavier aircraft require more complex and thus more heavy mechanical flight control system. To control the increased forces and reduce some weight hydraulic power was added to the control system. With hydraulic flight control systems, aircraft size and performance are limited by economics rather than a pilot's strength.

A hydromechanical flight control system has two parts:

- The *mechanical circuit*, which links the cockpit controls with the hydraulic circuits. Like the mechanical flight control system, it consists of rods, cables, pulleys, and sometimes chains.

- The *hydraulic circuit*, which has hydraulic pumps, reservoirs, filters, pipes, valves and actuators. The actuators are powered by the hydraulic pressure generated by the pumps in the hydraulic circuit. The actuators convert hydraulic pressure into control surface movements. The servo valves control the movement of the actuators.

2.1.3. Fly-by-Wire System

Fly-by-Wire is the generally accepted term for those flight control systems which use computers to process the control movements made by the pilot, or autopilot, and send appropriate electrical signals to the flight control surface actuators. In a fly-by-wire system a mechanical circuit from hydromechanical flight control system is substituted by an electrical circuit linking a cockpit controls to the hydraulic circuit.

The electrical circuit can be either analog or digital. In the analog fly-by-wire system the cockpit controls operate signal transducers which generate the appropriate commands, that are in turn processed by an electronic controller. The autopilot is a part of the electronic controller. In the digital fly-by-wire system digital computers are used. Unlike electronic controllers, digital computers can receive input from any aircraft sensor. They also increase electronic stability, because the system is less dependent on the values of critical electrical components in analog controllers.

The fly-by-wire system has several advantages. Amongst the most important ones the following belong:

- *Weight savings* - mechanical and hydro-mechanical flight control systems are heavy and require careful routing of flight control cables through the aircraft using systems of pulleys, cranks, tension cables and hydraulic pipes. Both systems often require redundant backup to deal with failures, which again increases weight.

- *Reliability improvement and safety* - the fly-by-wire system can respond flexibly to changing aerodynamic conditions by tailoring flight control surface movements so that aircraft response to control inputs is appropriate to flight conditions. It is also able to filter control inputs to prevent the pilot-induced oscilation, or can prevent dangerous fligh modes such as stalling or spinning. The electrical systems can be quadruplexed in order to prevent loss of signals in the case of failure of one or even two channels.

- *Maintenance* - mechanical and hydraulic systems require lubrication, tension adjustments, leak checks, fluid changes, etc., while electrical system requires less maintenance. The fly-by-wire system also introduces built-in-tests making regular checks of the systems easier.

2.2. Embedded System

An embedded system is a microprocessor-based system that is built to control a function or a range of functions and is not designed to be programmed by the end user. However, it can provide choices and different options for the user. Frequently, an embedded system is a component within a larger system often including hardware and mechanical parts.

Since the embedded system is dedicated to specific tasks, design engineers can optimize it to reduce the size and cost of the product and increase the reliability and performance.

Physically, embedded systems range from portable devices such as digital watches and MP3 players, to large stationary installations like traffic lights, factory controllers, or the systems controlling nuclear power plants. Complexity varies from low, with a single microcontroller chip, to very high with multiple units, peripherals and networks mounted inside a large chassis or enclosure.

2.3. Model Based Design

Model based design, or MDB, is a process that enables faster, more cost-effective development of dynamic systems, including control systems, signal processing, and communications systems. MDB is often used to complex control system, when on a project can work several engineers simultaneously. Model based design is a methodology often applied in designing software for embedded systems.

In model based design, a system model is at the center of the development process, from requirements development, through design, implementation, and testing. The model is an executable specification that is continually refined throughout the development process. After model development, simulation shows whether the model works correctly. This makes the development much quicker compared to the traditional control development methods. Using them the design takes place early in the development process, but test and implementation have to wait until late in the process. This delay is tied to the availability of production prototypes for testing the behavior of the control software.

For model-based design sophisticated tools and new development techniques exist. For example, model based control design software, such as National Instruments

MATRIXx or Matlab® Simulink®, and real-time hardware. The system is simulated and implemented earlier in the development cycle. These model based design tools have built-in mathematical functions and routines optimized for designing and analyzing control strategies through off-line simulation. This facilitates identifying problems and errors earlier in the process and thus reducing risk and development time and cost because of fewer iteration in the development cycle.

2.4. Honeywell Fly-by-Wire

Honeywell participates on development of several business and regional jets. Honeywell provides the flight control system, namely the fly-by-wire system. It is an embedded system and it is developped with a use of a model based design using Matlab® Simulink®.

3. Theoretical Background

In this chapter knowledge of graph theory which is direct or indirect foundation of this thesis is described. This chapter is based on [4],[6],[7].

3.1. Graphs

Definition 3.1.1 *Let V and E be disjoint sets and $\epsilon : E \to V \boxtimes V$* [1] *a mapping. A triple $G = (V, E, \epsilon)$ is called a **simple graph** G, the elements of V are called **vertices** and the elements of E are called **edges**. For edge $e = \{a, b\}$ a and b are called **end vertices**. We say that a and b are **incident** with e and that a and b are **adjacent** or **neighbours** of each other. Special case of edge $e = \{a, a\}$ is called a **loop**. A vertex which is incident with no edge is called an **isolated vertex**.*

Definition 3.1.2 *Let V and E be disjoint sets and $\epsilon : E \to V \times V$ a mapping. A triple $G = (V, E, \epsilon)$ is called a **simple directed graph**, or **simple digraph**, the elements of V are called **vertices** and the elements of E are called **edges** or **arcs**. Instead of $e = (a, b)$ we write $e = ab$ and a is called **start vertex** or **tail** and b is called **end vertex** or **head**. We say that a and b are incident with e, and call two edges of the form ab and ba **antiparallel**.*

Definition 3.1.3 *Let $G = (V, E, \epsilon)$ where there are at least two edges with the same end vertices $e_1 = \{a, b\}$ and $e_2 = \{a, b\}$, $a, b \in V$. Then G is called a **multigraph** and edges e_1 and e_2 are called **parallel**.*

Definition 3.1.4 *Let $G = (V, E, \epsilon)$ be a directed multigraph. Replacing each edge of the form (a, b) by an undirected edge $\{a, b\}$, we obtain the **underlying multigraph** $|G|$. Replacing parallel edges in $|G|$ by a single edge, we get the **underlying graph** (G). Replacing each edge $(a, b) \in E$ by two edges (a, b) and (b, a), we get the **associated directed multigraph** $\vec{G}$, or we also call $\vec{G}$ the **complete orientation** of G.*

Remark 3.1.5 *The complete orientation of G can be defined also for an unoriented multigraph $G = (V, E, \epsilon)$ by replacing each edge $\{a, b\} \in E$ by two oriented edges (a, b) and (b, a).*

Remark 3.1.6 *Further in this thesis we will understand a **graph** as a simple graph where there are no parallel edges. Analogously, we will understand **digraph** as a simple digraph.*

Definition 3.1.7 (Sets of vertices in digraphs)
Let $G(V, E, \epsilon)$ be a digraph, x, z its arbitrary vertices. Then we define following terms and notations:

[1] $V \boxtimes V$ is a set of all not ordered pairs $\{a, b\}, a, b \in V$

$V_G^+(x) = \{\forall z \in V : (x,z) \in \epsilon(E)\}$ *is a set of* ***descendants of a vertex x****, i.e. a set of all end vertices of edges with a start vertex* x*.*

$V_G^-(x) = \{\forall z \in V : (z,x) \in \epsilon(E)\}$ *is a set of* ***ancestors of a vertex x****, i.e. a set of all start vertices of edges with an end vertex* x*.*

$V_G(x) = V_G^+(x) \cup V_G^-(x)$ *is a set of* ***neighbours of a vertex x****, i.e. a set of all vertices connected to a vertex* x *by an edge.*

Example 3.1.8 *A directed graph is depicted in the Figure 3.1. The sets of vertices defined in the definition 3.1.7 for the vertex* f *are:*

$V_G^+(f) = \{g, h\}$

$V_G^-(f) = \{c, d, e\}$

$V_G(f) = \{c, d, e, g, h\}$.

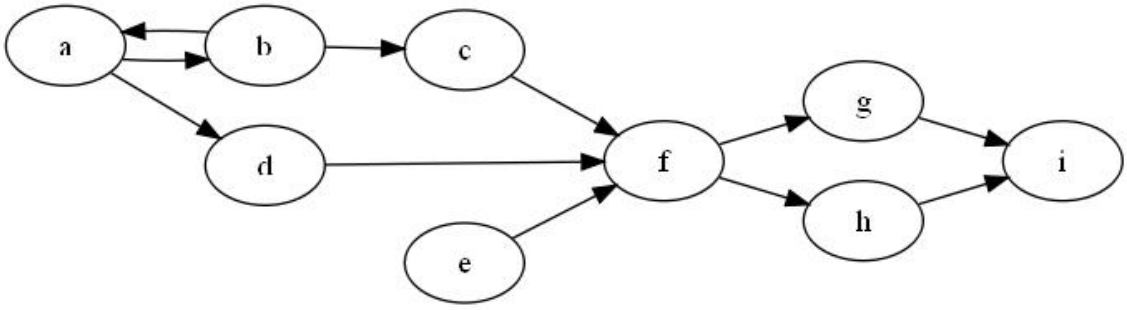


Figure 3.1: *A directed graph.*

Remark 3.1.9 *Set* $V_G(x)$ *can be redefined for graphs in a following way: Let* $G(V, E, \epsilon)$ *be a graph,* x, z *its arbitrary vertices. Then* $V_G = \{z \in V : \{x, z\} \in \epsilon(E)\}$ *is a set of neighbours of a vertex* x*, i.e. a set of all vertices connected to a vertex* x *by an edge.*

Definition 3.1.10 *Let* $G = (V, E, \epsilon)$ *and* $\bar{G} = (\bar{V}, \bar{E}, \bar{\epsilon})$ *both be graphs.*
If $\bar{V} \subseteq V$*,* $\bar{E} \subseteq E$ *and* $\bar{\epsilon}$ *is restriction of* ϵ*, graph* $\bar{G}$ *is called a* ***subgraph*** *of graph* G*. Analogously, subgraphs of digraphs and multigraphs are defined.*

Definition 3.1.11 *Let* $(e_1, \ldots, e_n)$ *be a sequence of edges in a graph* G*. If there are vertices* $v_0, \ldots, v_n$ *such that* $e_i = v_{i-1}v_i$ *for* $i = 1, \ldots, n$ *the sequence is called a* ***walk****. A walk can be also specified by a sequence of its vertices* $(v_0, \ldots, v_n)$ *provided that* $v_{i-1}v_i$ *is an edge* e_i *for* $i = 1, \ldots, n$*. A walk for which the* $e_i \neq e_j$ *for* $i, j = 1, \ldots, n$ *is called a* ***trail****. A trail for which* $v_i \neq v_j$ *for* $i, j = 0, \ldots, n$ *is called a* ***path****. If* $v_0 = v_n$ *the sequence is called a* ***closed walk, trail*** *or* ***path****, respectively.*

Definition 3.1.12 *A closed trail with* $n \geq 3$*, for which the* v_j *are distinct (except, of course,* $v_0 = v_n$*), is called a* ***cycle****.*

Definition 3.1.13 *A graph* $G = (V, R)$ *is* ***transitive*** *if relation* R *is transitive, it means* $[(x, y) \in R, (y, z) \in R] \Rightarrow (x, z) \in R$*.*

Definition 3.1.14 *A* ***transitive closure*** *of a graph* G *is the least transitive graph that contains graph* G *as a subgraph. It is marked* G^+*.*

3.2. Networks

Definition 3.2.1 *Let $G = (V, E, \epsilon)$ be a graph or a digraph on which there is a mapping $w : E \to \mathbb{R}$. We call the pair (G, w) a **network** and the number $w(e)$ is called the **weigth** or the **length** of the edge e.*

Remark 3.2.2 *For purposes of searching paths in a graph or a digraph we will assume that every graph or digraph is a network (G, w) where $w(e) = 1$ for every edge $e \in E$.*

Definition 3.2.3 *For each walk $W = (e_1, \ldots, e_n)$, the length of W is $w(W) := w(e_1) + \ldots + w(e_n)$.*
The length of a trivial path containing only one vertex is $w(W) := 0$.

Definition 3.2.4 *We define **distance** $d(a, b)$ between two vertices a and b in a graph or a digraph $G = (V, E, \epsilon)$ as follows:*

$$d(a,b) = \begin{cases} \infty \text{ if } b \text{ is not accesible from } a \\ \min\{\forall w(W) : W \text{ is a path from } a \text{ to } b\} \text{ otherwise} \end{cases}$$

*where $w(W)$ is length of a path W. Any path W (directed in the case of digraphs) for which the minimum in the equation is achieved is called a **shortest path** from a to b.*

Theorem 3.2.5 *Let (G, w) be a finite network. If there exists a path from vertex x to vertex y in the network, then there also exists the shortest path from vertex x to vertex y.*

Proof Number of edges in a graph is limited by number of its vertices. This means that there is a finite number of ways between x and y and therefore one of them has to be the shortest one.

Theorem 3.2.6 (Triangle inequality) *Let (G, w) be a network. If the network does not contain a cycle with a negative length, all triples of vertices of the graph satisfy inequality:*

$$d(x, z) \leq d(x, y) + d(y, z)$$

Proof For $d(x, y) = \infty$ or $d(y, z) = \infty$ the inequality is obvious. Let $d(x, y) < \infty$ and $d(y, z) < \infty$. By connecting the shortest path from a vertex x to a vertex y and the shortest path from a vertex y to a vertex z we obtain a walk from x to z. Either this way is a path or a path can be created from the way by omitting possible cycles. These cycles had a positive length, therefore the length of the path is at most $d(x, y) + d(y, z)$. At the same time the length of the path is at least $d(x, z)$.

4. Search Path Algorithms

In this chapter some algorithms which are able to find paths in a graph are described along with a way of measuring their quality.

We shall start with stating what an algorithm is or at least what should be understood when talking about an algorithm. An algorithm is a technique for solving problems. Here the term *problem* is used in a very general sense: it has a meaning of a *problem class* which consists of infinitely many *instances* having common characteristics. For example to search for a shortest path in a graph is a problem class while searching for a shortest path in a given graph G is an instance of the problem class.

According to the [6], an algorithm should have the following properties:

- *Finiteness of description*: The technique can be described by a finite text.
- *Effectiveness*: Each step of the technique has to be feasible (mechanically) in practice.
- *Termination*: The technique has to stop for each instance after a finite number of steps.
- *Determinism*: The sequence of steps has to be uniquely determined for each instance.

In most cases the last property is not required. Another important property is *correctness*, that is, it should indeed solve the problem correctly for each instance. Moreover, an algorithm should be *efficient*, which means it should work as fast and economically as possible. How the efficiency can be measured is described in the following section (Section 4.1).

There is a difference between an *algorithm* as described and a *program*. As an algorithm a general technique for solving a problem is understood (it means it is problem-oriented), while a program is the actual formulation of an algorithm as it is needed for being executed by a computer (and is therefore machine-oriented). Thus, the algorithm may be viewed as the essence of the program.

4.1. Complexity of Algorithms

Complexity theory studies the time and memory space an algorithm needs as a function of the size of the input data. Complexity is a way how to compare different algorithms which solve the same problem. For determining a complexity of an algorithm formally correctly, we would have to define what an algorithm is and define how to measure input data and time space. These can be defined with use of abstract computers, for example Turing machines, and postulating that certain operations are executed in a unit of time. However, reasonable good insight into complexity of algorithms is provided simply by an estimate for the resources needed by an algorithm. For measuring the size of input data for graph algorithms number of vertices and number of edges in the graph will be used.

The time complexity of an algorithm is a function f, where $f(n)$ is the maximal number of steps which the algorithm needs to solve a problem having input data of length n. The space complexity is defined analogously for the memory space needed. We do not specify what a step really is, but count the usual arithmetic operations, access to arrays, comparisons, etc. each as one step. The complexity of an algorithm is determined as a worst case scenario for given input data even though many algorithms are usually quicker than this estimate so it would seem better to determine something like average complexity. It would lead to need to determine probability distribution of input data and more complex and difficult calculation. Therefore it is common practise to look at the worst case scenario complexity.

The complexity $f(n)$ of an algorithm is generally not calculated exactly. Instead it is an estimation of how fast $f(n)$ grows. Following notation will be used throughout this thesis.

Let $f : \mathbb{N} \to \mathbb{R}_0^+$ and $g : \mathbb{N} \to \mathbb{R}_0^+$ be two mappings from set of natural numbers to set of nonnegative real numbers. We write

- $f(n) = O(g(n))$, if there is a constant $c > 0$ such that $f(n) \leq cg(n)$ for all sufficiently large n,
- $f(n) = \Omega(g(n))$, if there is a constant $c > 0$ such that $f(n) \geq cg(n)$ for all sufficiently large n,
- $f(n) = \Theta(g(n))$, if $f(n) = O(g(n))$ and $f(n) = \Omega(g(n))$.

If $f(n) = \Theta(g(n))$, we say that f has *rate of growth* $g(n)$. If $f(n) = O(g(n))$ then f has at most rate of growth $g(n)$, if $f(n) = \Omega(g(n))$ then f has at least rate of growth $g(n)$. If the time or space complexity of an algorithm is $O(g(n))$, we say that the algorithm has complexity $O(g(n))$.

We will usually consider the time complexity only and just talk about the complexity. Note that the space complexity is at most as large as the time complexity, because the data taking up memory space in the algorithm have to be read first.

4.2. Algorithms

As it was stated before, algorithms are techniques for solving problems. It is necessary to have rules how to describe them to avoid any ambiguity resulting from vague description. The ambiguity could lead even to a case when the algorithm would be misinterpreted and thus not solve the problem. Therefore following notation similar as in [6] is introduced:

Assigning value:

For assigning value a symbol := will be used. So $a := b$ means that a value b is assigned to a variable a.

Ramifications:

if *condition* **then**
  $P_1, \ldots, P_k$
else
  $Q_1, \ldots, Q_l$
end if

This is to be interpreted as follows: if *condition* is TRUE, the operations $P_1, \ldots, P_k$ are executed, and if *condition* is false, the operations $Q_1, \ldots, Q_l$ are executed. Here the alternative is optional so that we might also have

if *condition* **then**
  $P_1, \ldots, P_k$
end if

In this case, no operation is executed if *condition* is not satisfied.

Loops:

for $i = 1$ to n **do**
  $P_1, \ldots, P_k$
end for

specifies that the operations $P_1, \ldots, P_k$ are executed for each of the (integer) values the control variable i takes, namely for $i = 1, \ldots, i = n$. Other parameter for **for** loop can be

for $s \in S$ **do**
  $P_1, \ldots, P_k$
end for

which means that the operations $P_1, \ldots, P_k$ are executed $|S|$ times, once for each element s in S. The order of the elements, and hence order of the iterations, is not specified.

Iterations:

while *condition* **do**
$P_1, \ldots, P_k$
end while

This notation has the following meaning: if the *condition* is met (that is, if *condition* has Boolean value TRUE), the operations $P_1, \ldots, P_k$ are executed, and this is repeated as long as *condition* is met.

repeat
$P_1, \ldots, P_k$
until *condition*

requires first of all to execute the operations $P_1, \ldots, P_k$ and then, if *condition* is not yet satisfied, to repeat these operations until finally *condition* holds.

The main difference between describing iterations by **repeat until** and **while do** is that a **repeat** is executed at least once, whereas the operations in a **while** loop are possibly not executed at all, namely if *condition* is not satisfied.

4.3. Shortest Paths Problem

Let (G, w) be a network, x and y vertices of the graph and $d(x, y)$ a distance from a vertex x to a vertex y. The goal of the task is to find a shortest path or only its distance:

a) from a vertex x to a vertex y

b) from a vertex x to every other vertex of the graph

c) between every couple of vertices of the graph

In further text some solutions of tasks b) and c) will be described. The task a) is solved by algorithms used for b) and c) because there is no more useful way for solving the task.

If the task is to find shortest paths in a mutligraphs, it can be easily transfered to a task to find shortest paths in a graph by omitting all edges but the shortest of them.

Note that for all the mentioned tasks negative lengths of edges are allowed. However, dificulty of the task is strongly dependent on a fact whether a graph contains a cycle of a negative length. In this thesis algorithms for graphs without cycles of a negative length will be described. The following text is based on [4],[6] and [7].

4.3.1. Breath First Search Algorithm

Breath first search algorithm, or shortly BFS, is one of the most fundamental methods in algorithmic graph theory. BFS is an uninformed search method that examine all vertices of a graph or combination of sequences by systematically searching through every solution. In other words, it exhaustively searches the entire graph or sequence without considering the goal until it finds it.

Breath first search algorithm finds the shortest paths for unit-step costs, i.e. paths with the minimum count of edges, regardless of the kind of a graph, if the graph is finite. If the graph is infinite, the breath first search algorithm can diverge.

For the BFS algorithm a *queue* is used. It is a structure for which elements are always appended at the end, but removed at the beginning, it is also often called *FIFO* (first in – first out).

Algorithm 4.3.1 (BFS) Let G be a graph or digraph. Moreover, let s be an arbitrary vertex of G and Q a queue. The vertices of G are labelled with integers $d(v)$ as follows:

1: $Q := \emptyset$, $d(s) := 0$
2: append s to Q
3: **while** $Q \neq \emptyset$ **do**
4: remove the first vertex v from Q
5: **for** $u \in V_G^+(v)$ **do**
6: **if** $d(u)$ is undefined **then**
7: $d(u) := d(v) + 1$
8: append u to Q
9: **end if**
10: **end for**
11: **end while**

Theorem 4.3.1 *Algorithm 4.3.1 has complexity $O(|V| + |E|)$. At the end of the algorithm, every vertex t of G satisfies*

$$d(s,t) = \begin{cases} d(t) \text{ if } d(t) \text{ is defined} \\ \infty \text{ otherwise} \end{cases}$$

Proof In step 4: a vertex from the queue is read. Every vertex is stored in the queue only once - only if it is not processed. Thus the reading step will be performed $O(|V|)$ times, where $|V|$ is the number of the vertices in the graph. In a loop starting in step 5: all neighbours are examined or in other words all vertices of the currently read vertex are examined. Therefore the examining step will be performed $O(|E|)$ times, where $|E|$ is the number of the edges in the graph. Hence the complexity of BFS is $O(|V| + |E|)$. Moreover, $d(s,t) = \infty$ if and only if t is not accessible from s, and therefore $d(t)$ stays undefined throughout the algorithm. Now let t be a vertex such

that $d(s,t) \neq \infty$. Then $d(s,t) \leq d(t)$, since t was reached by a path of length $d(t)$ from s. We show that equality holds by using induction on $d(s,t)$. This is trivial for $d(s,t) = 0$, that is, $s = t$. Now assume $d(s,t) = n+1$ and let $(s, v_1, \ldots, v_n, t)$ be a shortest path from s to t. Then $(s, v_1, \ldots, v_n)$ is a shortest path from s to v_n and, by our induction hypothesis, $d(s, v_n) = n = d(v_n)$. Therefore $d(v_n) < d(t)$, and thus BFS deals with v_n before t during the while-loop. On the other hand, G contains the edge $v_n t$ so that BFS certainly reaches t when examining the adjacency list of v_n (if not earlier). This shows $d(t) \leq n+1$ and hence $d(t) = n+1$.

Note that algorithm 4.3.1 searches for shortest paths only for graphs without defined lenght of the edges - which is treated according to the Remark 3.2.2. It also only gives information about "how far a vertex is from the start vertex" but the actual path cannot be traced. Both these demerits solves the algorithm 4.3.2. It involves the following theory.

Basic scheme for computing a distance

For every vertex $x \in V$ a value $d(x)$, a length of the shortest path from a starting vertex s to the vertex x which has been found, is remembered. If no such path has been found, $d(x)$ is set to ∞. If the values d were equal to the real distances of vertices from the starting vertex s, the triangle inequality

$$d(x) + w(e_{xy}) \geq d(y)$$

should be satisfied for every edge from a vertex x to a vertex y. If the inequality is not satisfied, it means that the value $d(y)$ is not the length of the shortest path, because there is a shorter path containing the vertex x. Therefore the value $d(y)$ is decreased to

$$d(y) := d(x) + w(e_{xy}).$$

Testing the inequality for every couple of vertices and changing the value d if necessary is repeated as long as the inequality is satisfied for every edges of the network.

In a beginning of the calculation according to this scheme it is set $d(x) := \infty$ for every $x \in V \backslash \{s\}$ and $d(s) := 0$ because in the beginning no path from s but a trivial path has been found. A trivial path has a length $d = 0$ according to the definition 3.2.3.

Lemma 4.3.2 *Let (G, w) be a finite network without cycles of a negative length. Then at any time during the calculation of distances according the above mentioned scheme the distance of every vertex x from the starting vertex s is either $d(x) = \infty$ or $d(x)$ is the lenght of some path from s to x.*

Theorem 4.3.3 *Let (G, w) be a finite network without cycles of a negative length. Then the calculation according the above mentioned scheme ends after finite number of changes of a value d and after the calculation ends, $d(x) = d(s, x)$ for every vertex $x \in V$.*

The proofs of lemma 4.3.2 and theorem 4.3.3 can be found in [4], p. 63.

Algorithm 4.3.2 (Modified BFS) Let (G, w) be a network with no edges with negative length. Moreover, let s be an arbitrary vertex of G and Q a queue. $\pi(v)$ denotes the vertex immediately preceding v in the shortest path from s to v constructed by the algorithm. Also the vertices of G are labelled with integers $d(v)$ as follows:

```
1: Q := ∅, d(s) := 0, π(v) := ∅
2: append s to Q
3: for v ∈ V\{s} do
4:    d(v) := ∞
5: end for
6: while Q ≠ ∅ do
7:    remove the first vertex v from Q
8:    for u ∈ V_G^+(v) do
9:       if d(u) > d(v) + w(e_vu) then
10:         d(u) := d(v) + w(e_vu)
11:         π(u) := v
12:         append u to Q
13:      end if
14:   end for
15: end while
```

Theorem 4.3.4 *If the network does not contain a cycle of negative length, the algorithm 4.3.2 finds the shortest paths from a vertex s to other vertices of the graph.*

Proof Because of theorem 4.3.3 it is enough to show that the triangle inequality is satisfied for every edge of the graph. As it was stated before, the BFS algorithm exhaustively searches throughout the whole graph. For Modified BFS this feature remains. Therefore thanks to the line 9: where the trianle inequality is checked, we know that the inequality is satisfied for every edge.

4.3.2. Algorithm of Dijkstra

Dijkstra's algorithm, conceived by Dutch computer scientist Edsger Dijkstra in 1959, is a graph search algorithm that solves the single-source shortest path problem for a network with nonnegative edge path costs producing a shortest path tree. For a given source vertex s in the graph, the algorithm finds the shortest path between s and every other vertex. It can also be used for finding costs of shortest paths from a single vertex to a single destination vertex by stopping the algorithm once the shortest path to the destination vertex has been determined.

Algorithm 4.3.3 (Algorithm of Dijkstra) Let (G, w) be a network, where G is a graph or a digraph and all lengths $w(e)$ are nonnegative. $\pi(v)$ denotes the vertex immediately preceding v in the shortest path from s to v constructed by the algorithm. The algorithm calculates the distances with respect to a vertex s.

1: $T := V$, $d(s) := 0$, $\pi(v) := \emptyset$
2: **for** $v \in V \backslash \{s\}$ **do**
3: $\quad d(v) := \infty$
4: **end for**
5: **while** $T \neq \emptyset$ **do**
6: $\quad$ remove some u from T such that $d(u)$ is minimal
7: $\quad$ **for** $v \in V_G^+(u)$ **do**
8: $\quad\quad d(v) := \min\{d(v), d(u) + w(e_{uv})\}$
9: $\quad\quad$ **if** $d(v)$ was changed **then**
10: $\quad\quad\quad \pi(v) := u$
11: $\quad\quad$ **end if**
12: $\quad$ **end for**
13: **end while**

Theorem 4.3.5 *For every $v \in V$, the algorithm of Dijkstra finds a minimal path $W(s, v)$ and the distance $d(s, v)$.*

Proof We mark S a set of processed vertices, i.e. vertices removed from T. After first execution of the step 6 there is $S = \{s\}$ and the theorem is apparent.
Let's now assume that we are choosing a vertex u from T and that for a set S the theorem holds. It will be shown that for the vertex u the theorem holds as well.
Let's assume that the shortest path $W(s, u)$ is shorter than $d(u)$ at the time of choosing the vertex u. Let's mark y the first vertex in the path W which is not in S, and x the vertex preceding the vertex y in the path W. Because $x \in S$ and all edges leading to $V^+(x)$ are processed the equality $d(y) = d(s, y)$ is true. Since the lengths of edges are nonnegative, it has to be $d(s, y) \leq d(s, u)$. Also because the path W is shorter than $d(u)$, it has to be $d(s, u) < d(u)$, thus $d(y) < d(v)$. This is in a condradiction with the choosing criteria - choosing the vertext with the least value of d.

Theorem 4.3.6 *Algorithm 4.3.3 has complexity $O(|V|^2)$.*

Proof In step 6: the minimum of the $d(u)$ has to be calculated (for $v \in T$), which can be done with $|T| - 1$ comparisons. In the beginning of the algorithm, $|T| = |V|$, and then $|T|$ is decreased by 1 with each iteration. Thus we need $O(|V|^2)$ steps altogether for the execution of 6:. All other operations can also be done in $O(|V|^2)$ steps.

4.3.3. Algorithm of Floyd and Warshall

Algorithms in the previous text are all possible to use only for networks without edges with negative length. They also solve only a task of finding the shortest

paths from a certain starting vertex. However, sometimes it is useful to know the distances between every pair of vertices. Solving this task would result in following complexities:

$O(|V|(|V|+|E|))$: BFS algorithm

$O(|V|^3)$: algorithm of Dijkstra

These complexities are higher than original ones because for task to determine distances for every pair of vertices they need to be run $|V|$ times.

In this section, an algorithm for this problem is presented. It has the same complexity as the algorithm of Dijkstra for this problem, namely $O(|V|^3)$. However, it offers the advantage of allowing some lengths to be negative – although, of course, we cannot allow cycles of negative length. This algorithm determines the *distance matrix* $D = (d(v,w))_{v,w\in V}$ of the network.

Algorithm 4.3.4 (Algorithm of Floyd and Warshall) Let (G,w) be a network not containing any cycles of negative length and assume $V = \{1,\dots,n\}$. Put $w_{ij} = \infty$ if ij is not an edge in G.

```
1:  for i = 1 to n do
2:     for j = 1 to n do
3:        if i ≠ j then
4:           d(i,j) := w_ij
5:        else
6:           d(i,j) := 0
7:        end if
8:     end for
9:  end for
10: for k = 1 to n do
11:    for i = 1 to n do
12:       for j = 1 to n do
13:          d(i,j) = min{d(i,j), d(i,k) + d(k,j)}
14:       end for
15:    end for
16: end for
```

Theorem 4.3.7 *Algorithm 4.3.4 computes the distance matrix D for (G,w) with complexity $O(|V|^3)$.*

Proof The complexity of the algorithm is obvious. Let $D_0 = (d^0_{ij})$ denote the matrix defined on lines 3-7 of the algorithm and $D_k = (d^k_{ij})$ the matrix generated during the k-th iteration defined on line 13 of the algorithm. Then D_0 contains all lengths of paths consisting of one edge only. Using induction, it is easy to see that (d^k_{ij}) is the shortest length of a directed path from i to j containing only intermediate vertices from $\{1,\dots,k\}$. As we assumed that (G,w) does not contain any cycles of negative length, the assertion follows for $k = n$.

The distance matrix D constructed by the algorithm of Floyd and Washall 4.3.4 define a graph in a following way:

1. if $d_{ij} = 0$ there is a loop in the graph
2. if $d_{ij} = n$ where $n \neq 0$ and $n < \infty$ there is an edge with a length equal to n
3. if $d_{ij} = \infty$ there is no edge

From the main idea of the algorithm it is evident, that this newly created graph is a transitive closure for the original graph.

4.4. Depth First Search Algorithm

Depth First Search algorithm, or shortly DFS, is similarly as the Breath first search algoritm an uninformed algorithm for searching ways in graphs or digraphs, in other words it systematically searches through every possible solution until it searches through entire graph or finds the goal. However, while BFS searches for the shortests paths, DFS constructs the maximal paths.

DFS, progresses by expanding the first descentant vertex of the starting vertex that appears and thus going deeper and deeper until a goal vertex is found, or until it hits a vertex that has no descendants. Then the search backtracks, returning to the most recent vertex it has not finished exploring.

Similarly as the Breath first search algorithm, the Depth first search algorithm can diverge in infinite graphs. To assure that the algorithm terminates, it is often performed to a limited depth for very large or infinite graphs.

For implementation of the depth first search algorithm, a *stack* is often used. It is a structure for which elements are appended and removed from the end as well. It is also called *LIFO* (last in - first out). However, in the algorithm 4.4.1 other way of describing the algorithm is used, which allows to emphasize the backtracking step.

Algorithm 4.4.1 (DFS) Let G be a graph or a digraph. Moreover, let s be an arbitrary vertex of G. $\pi(v)$ denotes the vertex immediately preceding v in the path from s to v constructed by the algorithm. The edges of G are labelled with a boolean value $u(e)$ and vertices are labelled with integers $d(v)$ as follows:

1: **for** $v \in V$ **do**
2: $d(v) := 0$, $\pi(v) := \emptyset$
3: **end for**
4: **for** $e \in E$ **do**
5: $u(e) :=$ FALSE
6: **end for**
7: $v := s$
8: **repeat**
9: **while** there exists $w \in V_G^+(v)$ such that $u(vw) =$ FALSE **do**
10: choose some vertex $w \in V_G^+(v)$ with $u(vw) =$ FALSE
11: $u(vw) :=$ TRUE
12: **if** $d(w) = 0$ **then**
13: $d(w) := d(v) + 1$
14: $\pi(v) := w$
15: $v := w$
16: **end if**
17: **end while**
18: $v := \pi(v)$
19: **until** $v = s$ AND $u(sw) =$ TRUE for all $w \in V_G^+(s)$

Theorem 4.4.1 *Algorithm 4.4.1 has complexity $O(|V| + |E|)$ for connected graphs.*

Proof For connected graph depth first search algorithm visits every vertex in the graph and examines every its edge. Number of vertices is $|V|$ and number of edges is $|E|$. Therefore, DFS complexity is $O(|V| + |E|)$.

4.5. Representation of Graphs

As stated before (see section 4.2) algorithms are theoretical techniques for solving problems. However, to execute them in practice it is necessary to not only implement the algorithms in some programming language, but also requires to think about how to represent graphs, so the program can operate with them. In the further text a representation of digraphs is specified. For undirected graphs the same representations can be used, but the graphs have to be treated as their complete orientation.

Definition 4.5.1 Edge list.
A directed multigraph G on the vertex set $V = \{1, \ldots, n\}$ is specified by:

- *its number of vertices n,*
- *the list of its edges, given as a sequence of ordered pairs (a_i, b_i), that is, $e_i = (a_i, b_i)$.*

Edge list is an intuitive way how to represent graphs and it needs little space in memory, $2m$ places for m edges. However, it is not very convenient to work with, because to find all neighbours of a vertex v it is necessary to search through all the entire list which is time consuming. This is solved by the incidence list.

Definition 4.5.2 Incidence lists.
A directed multigraph G on the vertex set $V = \{1, \ldots, n\}$ is specified by:

- *the number of vertices n,*
- *n lists $A_1, \ldots, A_n$, where A_i contains the edges beginning in vertex i. Here an edge $e = ij$ is recorded by listing its name and its head j, that is, as the pair (e, j).*

Incidence lists are basically the same as the edge list, given in a different ordering and split up into n separate lists. Of course, in the undirected case, each edge occurs now in two of the incidence lists, whereas it would be sufficient to use the edge list where it is just once. But working with incidence lists is much easier, especially for finding all edges incident with a given vertex. If G is not a multigraph either directed or undirected, it is not necessary to label the edges, and the incidence lists degenerate into adjacency lists.

Definition 4.5.3 Adjacency lists.
A digraph G with vertex set $V = \{1, \ldots, n\}$ is specified by:

- *the number of vertices n,*
- *n lists $A_1, \ldots, A_n$, where A_i contains all vertices j for which G contains an edge (i, j).*

In the directed case, we sometimes need all edges with a given end vertex as well as all edges with a given start vertex, then it can be useful to store *backward adjacency lists*, where the end vertices are given, as well. For implementation, it is common to use ordinary or doubly linked lists. Then it is easy to work on all edges in a list consecutively, and to insert or remove edges.

Definition 4.5.4 Adjacency matrix
*A digraph G with vertex set $V = \{1, \ldots, n\}$ is specified by an $(n \times n)$-matrix $A = (a_{ij})$, where $a_{ij} = 1$ if and only if (i, j) is an edge of G, and $a_{ij} = 0$ otherwise. A is called the **adjacency matrix** of G.*

Adjacency matrix needs a lot of space in memory (n^2 places), it is not very convenient to use it unless to represent digraphs having many edges. Though adjacency matrices are of little practical interest, they are an important theoretical tool for studying digraphs.

5. Models

As it was mentioned in the chapter 2.4, Honeywell fly-by-wire system is mostly developed as a set of models implemented in Matlab® Simulink®. To make an orientation in the system easier, Honeywell engineers have developed several tools. One of them is FCM Graph which helps finding relations between models and trace signals flow. However, the smallest units that are displayed are models and signals as inputs or outputs of the models. Since it is desired to know besides an information that there is a relation between variables also how the signals are affected by inner model logic it is necessary to display also blocks on the way between an input and an output, therefore the information about blocks needs to be acquired from the models. In this chapter it is described how the models are processed to meet the requirement.

5.1. Model

For modeling, Simulink provides a graphical user interface which allows to build a model as a block diagram. Simulink includes a complex block library of sinks, sources, linear and nonlinear components, and connectors. It is also possible to customize and create your own blocks. After you define a model, you can simulate it, using a choice of mathematical methods.

5.2. Model versus Graph

Since a model is a block diagram one would intuitively identify blocks as vertices and lines as edges of a graph. Then it would be easy to implement common graph search path algoritms to find a relations of any blocks. However, it is not so simple. Because of parameters of individual blocks the model can be layered, one block can represent a set of blocks, a line can represent several lines, or there can be factual link where there is no line object in the block diagram. These are some of the reasons why it is necessary to create a graph based on the model and not to use the Simulink model itself as a graph. The final graph shall connect all factual links, flattens the model, splits up multiple lines and logic and highlights information about ports of blocks. The graph is printed into a text file in a Graphviz syntax. In further text it is described how the blocks of models are processed. All figures in this chapter are illustrational, the resulting graph is kept in memory and operated in its text form.

5.3. Blocks

Every block in a Simulink model have its set of parameters which determine its visual aspects and functionality. Some of the parameters are common for all Simulink blocks others are block-specific. They can be obtained by a Matlab function `getparam`.

For our purposes we determine only following block parameters which are then stored for every block in a structure `block_structures`:

- `Name` - every block needs to have a unique name at one level of a model. In any other level of the model, the name can repeat - and it often does for blocks named And, And1 etc.
- `MaskType` - indicates information about type of block, i.e. Inport, And, Switch etc.
- `Ports` - specifies information about number of inports, outports and enable, trigger, state, RConn, LConn and Ifaction ports. We are interested only in number of inports, outports and enable ports.
- `CompiledPortWidths` - gives the information about the width of signals flowing in or out of block. This parameter is available only for compiled model.
- `Parent` - returns name of the system that owns the block. If a block is in a lower level of model, the name is in a form of a path, eg. `model_name/subsystem_name`.

Because for some blocks signals flowing into different inports can have different influence to blocks' functionality we need to have an information about ports of the blocks to be able to accurately determine a relation between blocks. For this reason the information about ports is pointed out by creating vertexes for every single port for every block in the graph. We consider every outport being connected to every inport, therefore the block inner linkage is represented by all inport vertices being connected to a block vertex and so being outport vertices. There are some exception, for example see section 5.4.

In a graph every vertex needs to have unique specification. As stated above we cannot use a name, which would be intuitive choice, as a specifier since it is not unique even throughout a single model. Therefore a handle of a block is used. It is unique throughout a model. However, it does not need to be unique throughout a system of models, so every block is specified by a handle and a model name. Its ports are specified by the handle of the block and suffix `.in.number` or `.out.number` for inports or outports respectively. Parametr `number` specifies number of the port. They are numbered from 0. In Figure 5.1. there is an example for And block. The text representation of the graph looks as follows:

```
"4.118004e+003[0]_model01" [label="and[0]", shape="box"];
"4.118004e+003.in.0[0]_model01" [label="in.0[0]"];
"4.118004e+003.in.1[0]_model01" [label="in.1[0]"];
"4.118004e+003.out.0[0]_model01" [label="out.0[0]"];
```

where `4.118004e+003` is a handle of the block and parameter `[number]` specifies width of ports, respectively width of signals flowing in or out of the block. It goes

through 0 to width of the signal−1. See 5.4.2 for an example of signals wider than 1. These four lines of Graphviz code show example how the vertices are specified and how their attributes are set, more in ([12]).

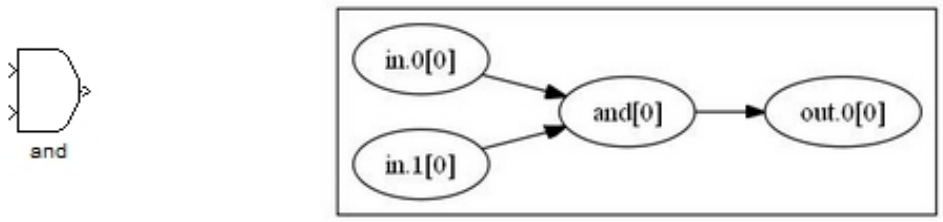


Figure 5.1: *Simulink And block and its representation in graph. For graph the information about ports of the block is stressed by creating vertices representing every port. Parameter [number] specifies width of ports, respectively width of signals flowing in and out of the block.*

Because every block in the graph is represented by its handle and name of the model it is part of, we need to have a way for matching blocks' names and their handles. For this purpose not only the graph is printed but also a lexicon file is printed. In the lexicon file there is a path to the block and its handle with other number parameters specified in text above. The path is created from block's parameter `Parent` and its name. For And block from example above the line in the lexicon file looks like this:

```
"model01/and" "4.118004e+003[0]_model01"
```

because it is a part of top level of model called model01. Note that in the lexicon file only blocks are printed, their ports are not.

5.4. Special Case Blocks

Some blocks have special features such as creating a connection while no link is in the model like blocks GOTO and FROM, such as being able to change a width of a signal or not having other function but making the model well-arranged. These blocks require extra attention while generating a graph based on a model. They are described in the following text.

5.4.1. Inports and Outports

Inports and Outports provide I/O for model or its subsystems. While other blocks are represented by their handle value and name of the model in the graph, I/O on model level has to be represented by their names to provide ability of connecting the models as well as graphs generated according to them. Therefore the program has to distinguish whether it is model I/O or a subsystem I/O.

In case of model I/O the program prints the information in the following way:

```
"inport[0]" [label="inport[0]", shape="box"];
"outport[0]" [label="outport[0]", shape="box"];
```

while for a subsytem I/O the program output looks like this:

```
"4.175004e+003[0]_model01" [label="inport_subs1[0]", shape="box"];
"4.258004e+003[0]_model01" [label="outport_subs1[0]", shape="box"];
```

where the model I/O signals are called "inport" and "outport" and the subsystem I/O signals are called "inport_subs1" and "outport_subs1". Parameter `[0]` added to both their names and handles is their width specification. Note that for Inport and Outport the port vertices are not printed since the blocks themselves are ports for models or subsystems.

5.4.2. Muxes, Demuxes and Logic of Muxed Signals

Mux block combines several input signals into a vector and Demux block extracts and outputs elements of a vector signal. Any logic between Mux and Demux blocks is evaluated as if it was implemented once for each component of the vectorized signal.

Intuitive way to deal with a muxed logic would be to print every block only once and a wider signal print as a multiple edge and thus create a multigraph. However, this solution is wrong because it would cause a mixing up signals that are not related to each other. For muxed logic every component of the vectorized signal is strictly separated from each other. Therefore it is necessary to evaluate blocks of muxed logic as multiple ones and print them width-of-signal times. So the Or block from Figure 5.3. is represented as follows:

```
"4.090002e+003[0]_model02" [label="or_sub[0]", shape="box"];
"4.090002e+003[1]_model02" [label="or_sub[1]", shape="box"];
"4.090002e+003[2]_model02" [label="or_sub[2]", shape="box"];
"4.090002e+003.in.0[0]_model02" [label="in.0[0]"];
"4.090002e+003.in.0[1]_model02" [label="in.0[1]"];
"4.090002e+003.in.0[2]_model02" [label="in.0[2]"];
"4.090002e+003.in.1[0]_model02" [label="in.0[0]"];
"4.090002e+003.in.1[1]_model02" [label="in.0[1]"];
"4.090002e+003.in.1[2]_model02" [label="in.0[2]"];
"4.090002e+003.out.0[0]_model02" [label="out.0[0]"];
"4.090002e+003.out.0[1]_model02" [label="out.0[1]"];
"4.090002e+003.out.0[2]_model02" [label="out.0[2]"];
```

which you can see in a Figure 5.4.

Mux and Demux blocks need special attention. As stated before all components of muxed signals are strictly separated and since Mux blocks compose these signals and Demux blocks decompose these signals it is evident that these blocks cannot be treated as the rest of blocks where all inports are connected to every outport. On the contrary, the outport of Mux block and Mux blocks itself need to be printed width-of-output-signal times so the resulting connection is following: first inport vertex is connected to first Mux block vertex which is connected to first outport vertex and so on for others inports of the Mux block. Analogously it is done for Demux block, only the extended port is inport. See Figure 5.4.

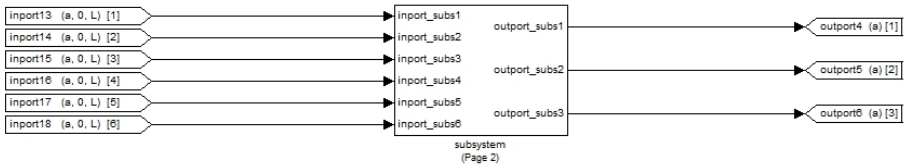


Figure 5.2: *Top level of a model.*

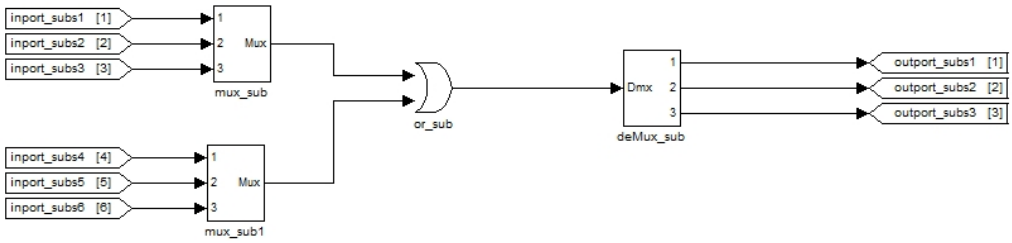


Figure 5.3: *Subsystem level of the Figure 5.2. model. The input signals are muxed into two vectorized signals of a width 3. Therefore the Or block is evaluated as 3 separate Or blocks. The resulting ored vectorized signal is then decomposed into its components.*

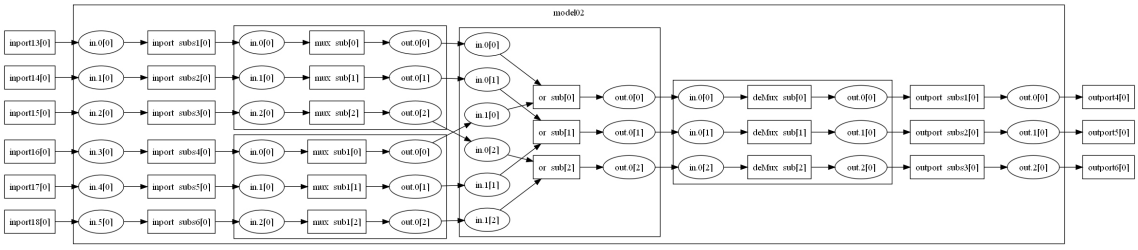


Figure 5.4: *Graph based on model pictured in Figure 5.2. Vertices "in.0[0]", "in.1[0]" ... and "out.0[0]", "out.1[0]" and "out.2[0]" which are not encased in any cluster are ports of the subsystem. Each of the Mux blocks vectorizes 3 signals and therefore the Or block is triplicated.*

Muxed logic requires processing of signals of the same width, i.e. if any block of muxed logic has one input signal of width n any other input signal of the block

needs to have a width n otherwise Simulink reports an error. However, there is one exception for this rule. Simulink is able to process simple signal entering a muxed logic by expanding it to appropriate width so it affects every component of the muxed signal. This is often used for switching signals, reset signals, constant signals and so on. Therefore also in the graph a simple signal entering a muxed logic is expanded as is demonstrated in Figure 5.5.

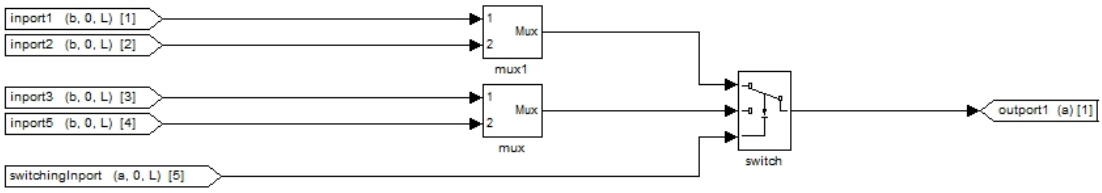


Figure 5.5: *The muxed signals are switched in accordance with a value of boolean switching signal. The switching signal has a width 1 but it is evaluated for every component of the muxed signals.*

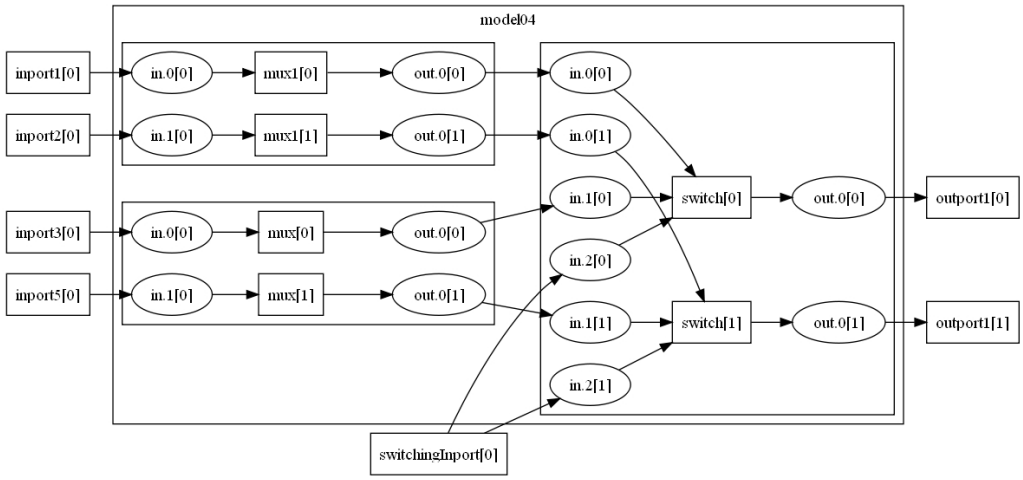


Figure 5.6: *Graph based on model pictured in Figure 5.5. Since the Switch block is a part of a muxed logic with signal width 2, it is printed twice. The switching signal "switchingInport" is not expanded in a way of muxed logic, it does not have multiple vertices each one for each component, but it is connected to both switching inports "in.2[0]" and "in.2[1]" of the Switch block.*

5.4.3. Subsystems

A Simulink model can consist of layers. Each layer is defined by a subsystem. A subsystem is part of the overall model and has no impact on the meaning of the model. Subsystems are provided to help in the organization of aspects of the model. Since a subsystem has no functionality of its own there is no need to keep the information of its use in the model in the graph. However, it can be useful to keep the information about a use of subsystem in the model so the engineer can locate a block found by any search algorithm more easily. For that reason subsystems are involved in the graph as their ports. See Figure 5.4. or Figure 5.9. for example.

5.4.4. Enable block

Use of enable block in a subsystem adds ability to enable or disable logic encapsulated in the subsystem. An enabled subsystem executes while the input received at the enable port is greater than zero. When an enabled subsystem is disabled, the subsystem output signals are either reset to zero or their last value is held. It is decided in accordance with a responsible parameter of the Enable block setting. Therefore the subsystem output signals are affected by the enabling signal even though there are no links between an enable block and output signals. The program reflects this fact and adds the links to the graph. In case that a subsystem output is a muxed signal, the link is added to each element of the vectorized signal. See Figure 5.9.

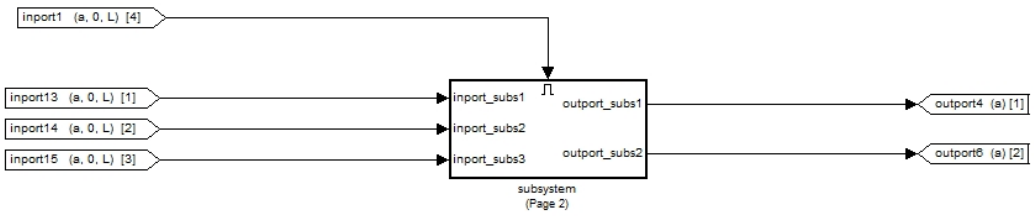


Figure 5.7: *Top level of a model containing only one enable subsystem. "inport1" is enabling signal.*

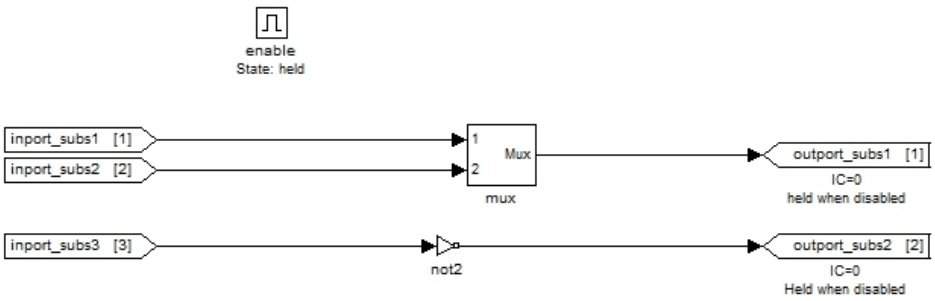


Figure 5.8: *Inner logic of enabled subsystem. Enable block is connected with no other block. The outport signals are linked with the enable block by setting value of responsible parameter. In the picture you can see that the value of the parameter is set to "held when disabled" with initial condition (IC) equal to zero which is used before the subsystem is enabled for the first time. When subsystem has been enabled the last value will be held after disabling the subsystem.*

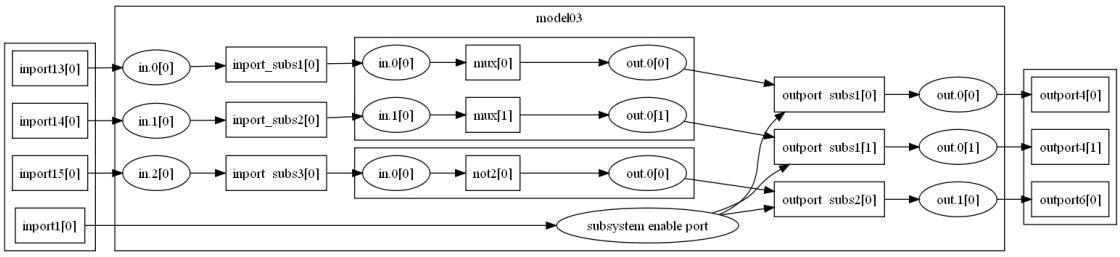


Figure 5.9: *Graph based on model pictured in Figure 5.7. "inport1" in enabling signal. It flows into enable port of subsystem. The enable port is connected to subsystem output signal and every component of muxed output signal as well.*

5.4.5. Stateflow

In Simulink, a Stateflow block uses a Stateflow diagram to represent an object with a discrete set of modes. These modes are known as states. States are connected by transitions which are often conditioned. Stateflow diagram represents if-else programming. Relations of inputs and outputs of Stateflow block can be very complex and creating a graph representing them is very complicated. Therefore we consider being sufficient to say that every input of the Stateflow block is related to its every output. See Figures 5.10. - 5.13.

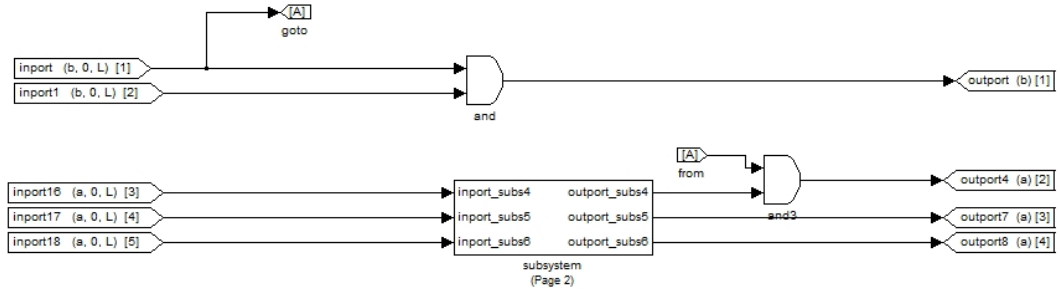


Figure 5.10: *Top level of a model with subsystem and blocks Goto and From with a tag [A].*

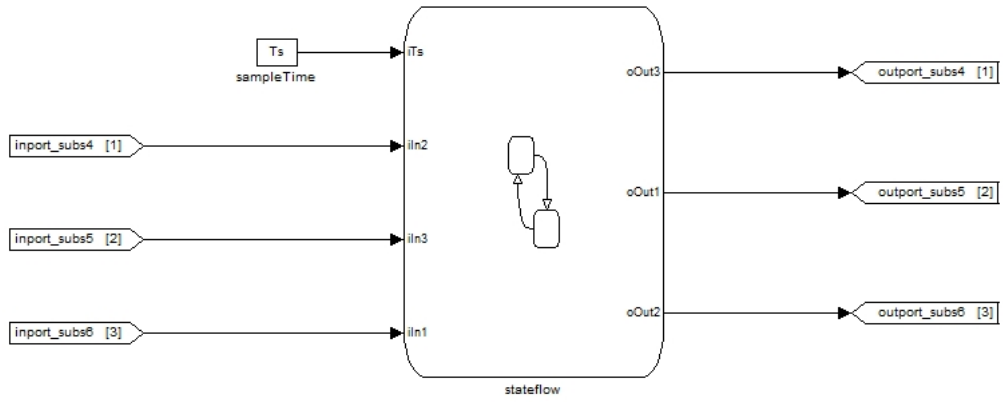


Figure 5.11: *Subsystem logic containing Sampling Time block and Stateflow block.*

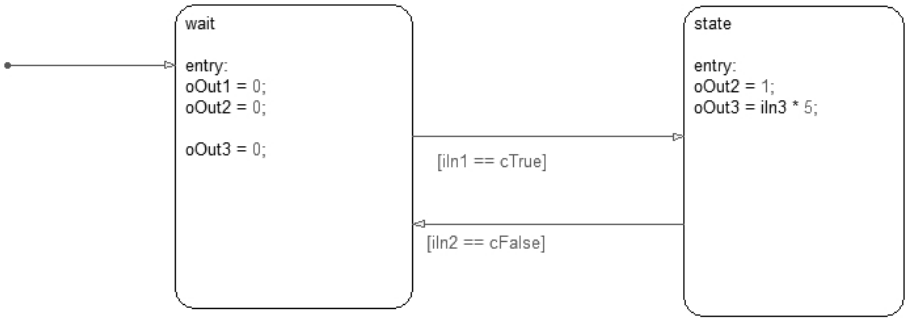


Figure 5.12: *Stateflow diagram containing only two states. In the first state - "wait" state output signals "oOut1", "oOut2" and "oOut3" are set to 0. When an input signal iIn1 meets condition iIn1 == cTrue, it is transitioned to second state "state" where signals "oOut2" and "oOut3" are set. If input signal iIn2 meets condition iIn2 == cFalse it transitions back to "wait" state.*

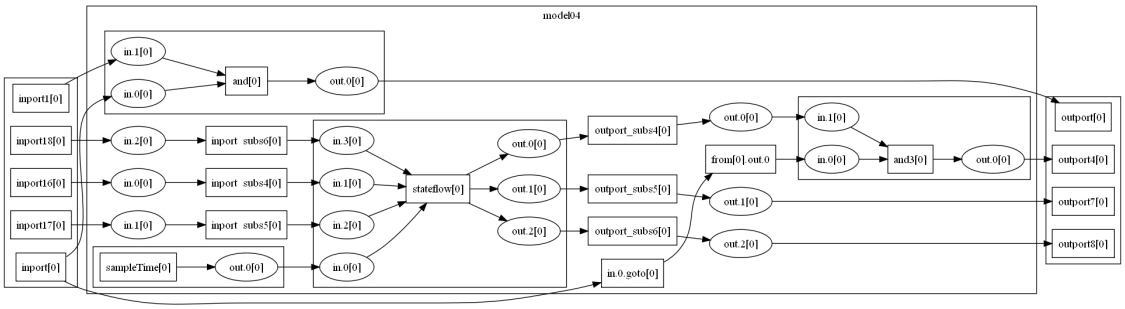


Figure 5.13: *Graph based on model pictured in Figure 5.10. Stateflow logic is reduced to every input being related to every output. Blocks Goto and From are connected by an edge.*

5.4.6. Blocks Goto and From

The Goto block passes its input to its corresponding From blocks. The input can be also vectorized signal of any data type. From and Goto blocks allow you to pass a signal from one block to another without actually connecting them. A Goto block can pass its input signal to more than one From block, although a From block can receive a signal from only one Goto block. The input to that Goto block is passed to the From blocks associated with it as though the blocks were physically connected. Therefore in the graph the link is added, see Figure 5.13. For Goto and From blocks the port vertices are not printed as well as for Inport and Outport blocks.

5.4.7. Selector Block

The Selector block outputs selected or reordered elements of input signal which can be vector, matrix, or multidimensional signal. In Honeywell only vector input is allowed. The output signal is specified by the user by setting parameter `Elements`

equal to a vector containing any variation of numbers of input vector components' order. By length of the Elements vector the width of output signal is specified. Some components of the input signal does not need to be part of the output signal and thus be terminated by the Selector block. See Figure 5.15.

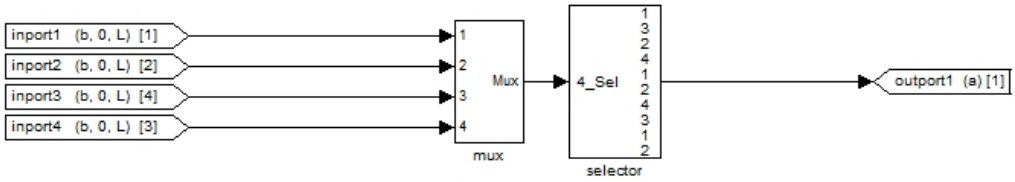


Figure 5.14: *Model containing Selector block which inputs muxed signal of width 4. The output signal is vectorized signal as well, its width is 10. Its components are reorganized components of original signal, they are in a following order:* $[1, 3, 2, 4, 1, 2, 4, 3, 1, 2]$.

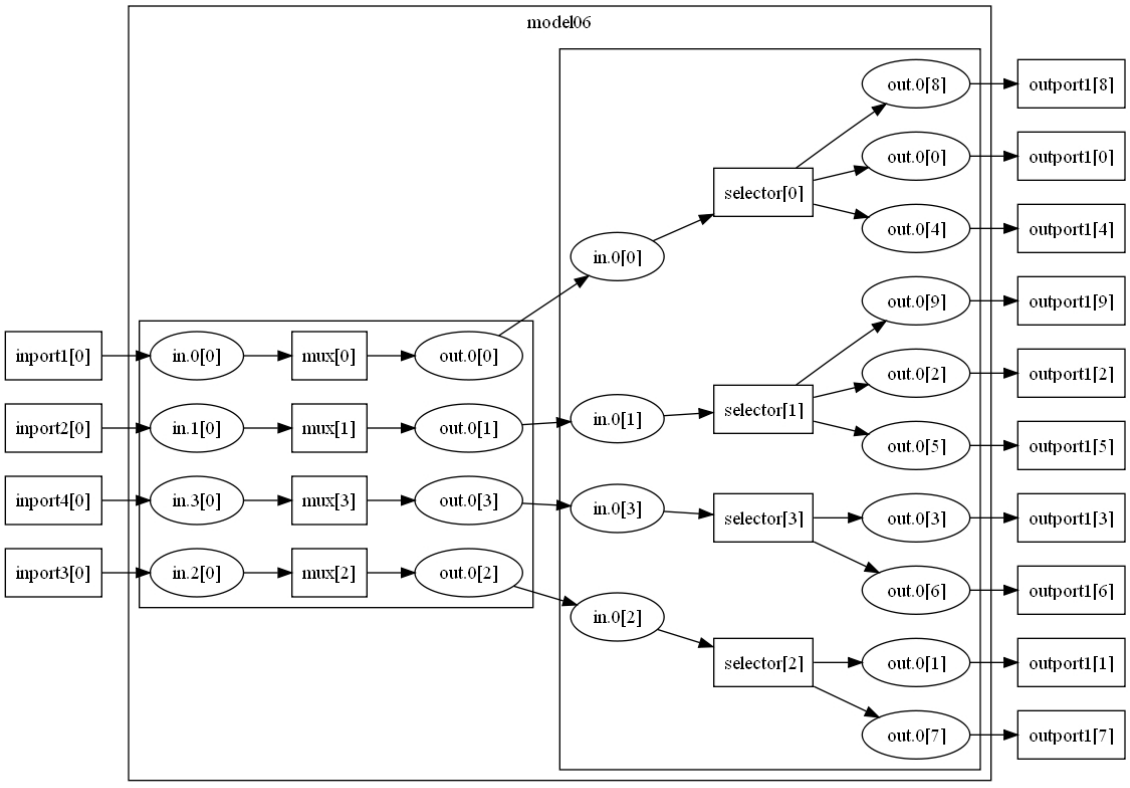


Figure 5.15: *Graph based on model pictured in Figure 5.14. The Selector outputs signal of width 10. Each component of the output signal is connected only to the appropriate component of the input signal.*

5.5. Edges

In the previous text the declaration of vertices of the graph is described. The edges should be considered now. To describe the model correctly, it is essential to keep information about data flow, therefore the edges have to be directed. In the following text the detection and syntax of edges are described. There are three different types of links in the models, each of them require its specific approach to model them by the edges. They are blocks' inner connections, blocks' connections and virtual connections.

5.5.1. Blocks' Inner Connections

As it was stated before, every block is represented by several vertices. One vertex represents the block itself and other vertices represent the block's inports and outports. To model the fact that every outport of a block is connected to every inport of the block, there exists an edge from every inport vertex to the block vertex and an edge from the block vertex to every outport vertex, as it is illustrated for example in Figure 5.1 and even better in Figure 5.13 for the Stateflow block.

The blocks' inner connections are constructed according to parameters stored in the structure `block_structures`, specifically the `Ports` parameter. The edges are printed to the same file as vertices, and a Graphviz syntax is used as well. For example edges in the And block from Figure 5.1 would be defined in a following way:

```
"4.118004e+003.in.0[0]_model01" -> "4.118004e+003[0]_model01" [len=1];
"4.118004e+003.in.1[0]_model01" -> "4.118004e+003[0]_model01" [len=1];
"4.118004e+003[0]_model01" -> "4.118004e+003.out.0[0]_model01" [len=1];
```

In this sytnax the vertex standing before "`->`" is the start vertex of the edge, the vertex standing after "`->`" is the end vertex of the edge. `[len=1]` is a parameter specifying a weight of the edge. More information about the weight parameter is in section 5.5.4.

5.5.2. Blocks' Connections

As it was mentioned in a section 5.2, intuitively one would create edges of the graph according to the lines in the original model. It would make some problems for muxed logic, but it could be dealt with. However, there is a reason, why the lines cannot be used. The reason is that lines do not have necessary properties to allow us to obtain required details, such as which blocks it connects or even which their ports are connected. Therefore other approach needs to be used.

Every block "knows" about its neighbours. The parameter in which the links of the blocks are stored is called `PortConnectivity`. This parameter can be obtained by function `getparam`. The line of a code has the following syntax:

```
getparam(block_handle,'PortConnectivity')
```

This parameter is an array of structures each of which describes one of the block's input or output ports. Each port structure has the following fields (see Matlab help [8]):

- `Type` specifies the port's type and/or number.
- `Position` stores a two-element vector, $[x, y]$, that specifies the port's position.
- `SrcBlock` specifies the handle of the block connected to this port. This field is null for output ports and -1 for unconnected input ports.
- `SrcPort` specifies the number of the port connected to this port, starting at zero. This field is null for both output ports and unconnected input ports.
- `DstBlock` specifies the handle of the block to which this port is connected. This field is null for input ports and contains a 1-by-0 empty matrix for unconnected output ports.
- `DstPort` specifies the number of the port to which this port is connected, starting at zero. This field is null for input ports and contains a 1-by-0 empty matrix for unconnected output ports.

For the program only `DstBlock` and `DstPort` fields are used because the script goes through every block and otherwise the edges would be doubled. In this way every block, apart from Outport blocks, has its descendant. These couples of blocks with information about ports are stored in a structure and then printed to a .txt file in a graphviz syntax. It was decided to use `DstBlock` and `DstPort` because it better shows the flow of signals.

5.5.3. Virtual Connections

In section 5.4.3 it was said that subsystems are used to make a model easier to understand. Every subsystem has inner Inport and Outport blocks. There are virtual connections of the subsystem's ports and its inner Inport and Outport blocks. This virtual connections have to be substituted by an edge in the graph.

For matching a subsystem's ports to appropriate Inport or Outport blocks, it is necessary to have a list of the Inport and Outport blocks. The list is obtained by following commands:

```
find_system(subsystem_handle,'FindAll','on','SearchDepth',1,
     'blocktype','Inport')
find_system(subsystem_handle,'FindAll','on','SearchDepth',1,
      'blocktype','Outport')
```

It is also necessary to have the information about which port of the subsystem is connected to which Inport or Outport block. The Inport and Outport blocks are ordered within the subsystem and their numerical order corresponds to the number of the subsystem's port which they are linked to. The number of the order of the Inport or Outport blocks is in the parameter `Port`. As it was stated before, subsystems are represented only by their port vertices and Inports and Outports are represented only by their block vertices. Therefore the edges connecting the subsystem with its inner Inports and Outports have one end vertex a subsystem's port vertex and the other end vertex an Inport or Outport block vertex.

Other blocks with a virtual connection are Goto and From blocks. Because one Goto block can pass a signal to several From blocks, all of them linked to the Goto block by a parameter `GotoTag`, the couple of Goto and From blocks are determined in a following way:

```
gotoTag = getparam(from_handle,'GotoTag');
goto_handle = find_system(parent,'FindAll','on','SearchDepth',1,
      'blocktype','Goto','GotoTag',gotoTag);
```

where `parent` is a handle of the model's level where the Goto and From blocks are.

5.5.4. Weight of Edges in the Graph

The main purpose for creating the graph according to the models is to be able to search for paths from a block to some another and determine whether it is a primary or a secondary path. To be able to decide which of them the path is, it is necessary to involve a weight of edges. In this section, the weights are described.

The weight of edges should reflect importance of blocks' ports in original models. Here the words "importance of ports" mean how an input signal is reflected in an output signal of the block. If the output signal is affected by an input signal directly, i.e. the data transmitted by the input signal are direct part of the output signal even though processed by the block's functionality, the input signal is considered to flow to a primary port. If the output signal is affected by an input signal indirectly, i.e. the data transmitted by the input signal affects the way how other input signals are processed by the block's functionality, the input signal is considered to flow to a secondary port.

There are several options how to set the weight to the edges. The weight can be assigned to blocks' connecting edges, to blocks' inner connecting edges, or to both

types of edges in accordance with which vertices, or better say ports, they connect. The idea of distinguishing the primary and secondary paths is that when a path contains any secondary port, it is secondary, otherwise it is primary. Therefore there is no need to assign weight to both types of edges.

For practical programming reason it was decided to set the weight of blocks' connecting edges equal to 1 and set the decisive values to blocks' inner connecting edges. Weight equal to 1 is set to edges with both end vertices of primary ports, and weight equal to 2 is set to edges having either the start or the end vertex of a secondary port.

The question remains how to determine primary and secondary ports. As the easiest way emerged to hardcode the port importance to every block. It is possible because Honeywell engineers use limited library of blocks while creating models. They can use about 150 blocks. In cooperation with experienced Honeywell engineers the port importance was determined for every single block according to their engineering judgement.

6. Computer Program

The computer programs implementing described knowledge are integral part of this thesis. In this chapter the programming languages used for implementation of the programs are described, the results are discussed and a brief manual for use of the programs is presented.

6.1. Programming Languages

The program consists of two parts. One is written in Matlab. It is a program which creates the graph according to models as it is described in chapter 5. The other part which is responsible for searching paths in the graph is written in Perl. Both these parts use syntax of Graphviz. All the three mentioned programs or programming languages are described in this section.

6.1.1. Graphviz - Graph Visualisation Software

Graph Visualisation Software, or shortly Graphviz, is a platform for graphical representation and manipulation with graphs and networks. It is an open source software licensed under the Common Public License. Graphviz has several graph layout programs, web and interactive graphical interfaces, and auxiliary tools, libraries, and language bindings.

In the program the **dot** layout language is used. It works with directed graphs and uses algorithm which aims edges in the same direction and then attempts to avoid edge crossings and reduce edge length.

More information and instalation file for Graphviz and dot language can be found on Graphviz website, see [12].

6.1.2. MATLAB

MATLAB® developped by The MathWorks is a high-level language and interactive environment that enables you to perform computationally intensive tasks faster than with traditional programming languages. The name Matlab stands for "Matrix Laboratory". MATLAB is the foundation for Simulink® and all other MathWorks products, and can be extended with add-on products for optimization, statistics, control systems, etc. More information on the official website of The MathWorks [13].

6.1.3. Perl

Perl is a general-purpose programming language originally developed for text manipulation and now used for a wide range of tasks including system administration, web development, network programming, GUI development, and more. The language is

intended to be practical (easy to use, efficient, complete) rather than beautiful (tiny, elegant, minimal). Its major features are that it's easy to use, supports both procedural and object-oriented (OO) programming, has powerful built-in support for text processing, and has one of the world's most impressive collections of third-party modules. Perl is a General Public Licence software. More information about Perl and its instalation files can be found at official website [14].

6.2. Searching Paths

In chapter 4 some search path algorithms are described. They all use the definition 3.2.3 of length of a path. However, for problem of determining if the found path is primary or secondary, this way of determining the length of path is not very suitable. The Honeywell fly-by-wire software consists of many models, so the paths connecting two vertices can be very long. If the lenght was calculated as a sum of weights of all edges of the path, the weights would need to be set very carefully to make sure that a long primary path does not appear to be longer than a short secondary path. The deciding criteria would need to be set to ensure that no primary path would be considered to be a secondary path and vice versa. Rather difficult and tiresome analysis would need to be done to set the weights and the treshold correctly. However, there exists an elegant solution for the problem of finding the reliable deciding criteria. The solution is to define the lenght of a path in other way.

Definition 6.2.1 *For each walk* $W = (e_1, \ldots, e_n)$*, the length of* W *is* $w(W) := \max\{w(e_1), \ldots, w(e_n)\}$.
The length of a trivial path containing only one vertex is $w(W) := 0$.

With this definition of a length of a path and with setting weights of edges as described in section 5.5.4, the primary path has always a length equal to 1 and secondary path has always a length equal to 2 no matter how many edges the path contains, considering the weights set as described in chapter 5.5.4.

The search path algorithms are then modified in the following way:

BFS:

```
    ⋮
  if d(u) is undefined then
    d(u) := max{d(u), w(e_uv)}
    π(u) := v
    append u to Q
  end if
    ⋮
```

Algorithm of Dijkstra:

```
    ⋮
for v ∈ V_G^+(u) do
    d(v) := min{d(v), max{d(u), w(e_uv)}}
    if d(v) was changed then
        π(v) := u
    end if
end for
    ⋮
```

Algorithm of Floyd and Warshall:

```
    ⋮
for k = 1 to n do
    for i = 1 to n do
        for j = 1 to n do
            d(i, j) = min{d(i, j), max{d(i, k), d(k, j)}}
        end for
    end for
end for
    ⋮
```

DFS:

```
    ⋮
if d(w) = 0 then
    d(w) := max{d(v), w(e_vw)}
    π(v) := w
    v := w
end if
    ⋮
```

6.3. Results

The algorithms modified to distinguish primary and secondary paths were tested on set of Honeywell data. The data are confidential therefore only discusion of the results is introduces and not the results themselves.

6.3.1. Algorithm of Floyd and Warshall

The algorithm of Floyd and Warshall determines a distance matrix of the network with complexity $O(|V|^3)$. Even though the complexity of the algorithm is high, it might be still be useful to know if a path exists between two vertices. There is

a presumption that constructing the distance matrix by using the algorithm of Floyd and Warshall will take a long time, but if it is reasonable long time it still might be rewarding in saving time in further use, if we were able to tell immediately if there is no path between two vertices. The algorithm was implemented and we obtained the following results:

Number of models: 5
Number of vertices: 8974
Duration of one cycle: 0,00000646311566748384 s
Number of cycles: $3 * 8974 = 722700234424$
Total time: 4 670 895 s $\approx$ 54 days

The time of constructing the distance matrix can be reduced by implementing the algorithm in some other programming language, for example C++, but the reduction would not be significant. Considering that the fly-by-wire system consists of approximately 1250 models, the algorithm of Floyd and Warshall is definitely not suitable for this problem.

6.3.2. Modified BFS, DFS, Algorithm of Dijkstra

The goal of the search path algorithms is not to find the shortest or the longest paths. The goal is to find a path connecting two vertices and to decide if it is a primary or a secondary path. For this reason it is possible to use Modified BFS, Dijkstra's and also DFS algorithm because all of them exhaustingly search through all the graph.

All the three algorithms were used to find paths between the same couples of vertices. With the modified length calculation, BFS and algorithm of Dijkstra find the same paths, while DFS finds different paths. This result was expected. More interesting is a difference in time needed by an algorithm to find the path. With a respect to complexities of the algorithms, to remind:
BFS: $O(|V| + |E|)$
DFS: $O(|V| + |E|)$
Dijkstra: $O(|V|^2)$
it could be expected that BFS and DFS would take similar time to find the path while algorithm of Dijkstra would take longer. However, the results show that DFS takes longer time than BFS to find the path. Dijkstra's algorithm takes longer time than BFS entirely according to the expectations.

To mention one particular example to illustrate the time consumed by the algorithms, the algorithms were supposed to find a path between signals
`norm_mode_eng_csol` and `cas_filt`. All of them found a secondary path.

BFS found a path containing 52 vertices in 1 second.
Algorithm of Dijkstra found the same path in 4 seconds.
DFS found a path containing 85 vertices in 4 seconds.

It seems that BFS is most suitable for problem of searching primary or secondary paths. However, it depends on what an engineer needs to find out. If they are interested merely in a fact if there exists a path between two blocks, then the use of BFS algorithm is advisable. For other applications the comparison of paths found by BFS and DFS can be useful. Only the algorithm of Dijkstra is not suitable for this problem, because it finds the same paths as BFS but in significantly longer time.

6.4. Program Manual

The Matlab script consists of three functions `batch_parser.m`, `parser.m` and `port_importance.m`. The controlling function called by user is `batch_parser`, which for every model calles `parser` which returns list of all blocks in the model with their parameters mentioned in section 5.3 and list of couples of connected blocks. The function `parser` calles `port_importance` for every block to determine importances for the block's ports.

The function `batch_parser` is called by user from the Matlab command window in the following way:

```
batch_parser(inp_folder)
```

where `inp_folder` is a path to a folder with models which are to be processed. `batch_parser` creates a subfolder `output` where output files - `model_name.dot` and `model_name_lexicon.txt` are stored.

Honeywell uses its own add-on for Matlab which limits some features of Simulink to provide ability of generating a standardized C++ code out of the models. The script `batch_parser` is possible to use only with this add-on, otherwise an error is reported.

The program `find_micro_path.exe` is a console application. It has two compulsory parameters and three optional parameters. If the optional parameters are not set, their default value is used.

The compulsory parameters are *Start vertex* and *End vertex* which are specified by their lexicon names.

The optional parameters are:

- `-alg=` specifies which algorithm is used to search for paths. Possible choices are BFS, DFS and Algorithm of Dijksta. Default algorithm is BFS. To change algorithm parameter this syntax is used: `-alg=BFS` for the BFS algorithm, `-alg=DFS` for the DFS algorithm and `-alg=Dijkstra` for the algorithm of Dijkstra.
- `-ot=` specifies type of output. The program either outputs a text description of the paths, `-ot=txt`, or creates an image with a use of Graphviz, `-ot=dot`. Text description is the default.
- `-o=` specifies output file of the program. If the output type is set to `txt`, this parameter does not need to be set, then the program prints the path to the console. If the parameter is set, `-o=file_name.txt`, the program creates the text file to which the path is printed. If the output type is set to `dot`, the output file has to be specified, i.e. `-o=file_name.png`.

To run the program `find_micro_path.exe`, the files generated by the Matlab script containing the graph structure have to be in a folder called `data` inside the folder containing the program itself.

The text output of the function is in the following format:

```
Elapsed time:  n seconds

1(w):  Start_vertex - vertex1 - vertex2 - ...- vertexN - End_vertex
```

where `(w)` is the weight of the path. Its value can be either 1 for a primary path or 2 for a secondary path. The number `1` in the example is a number of path. The paths are numbered to distinguish them in case there are several paths found.

The image output of the function looks in the following way:

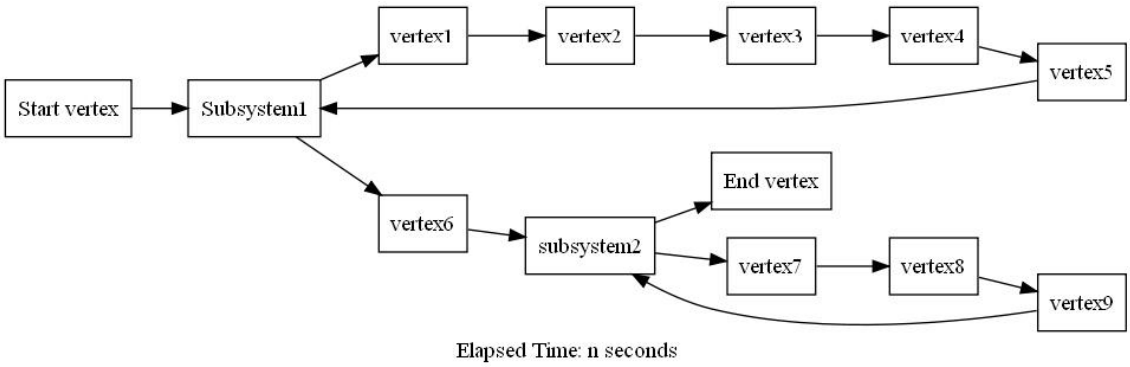


Figure 6.1: *The illustration of image output of program* `find_micro_path`.

It might seem redundant to generate an image of a path, which is a sequence of vertices and edges. As you can see in Figure 6.1, there are cycles, which must not appear in a path. Otherwise it would not be a path but a trail. However, the cycles are not cycles in the original graph. The image is generated in a way to highlight logic that is encapsulated in a subsystems. So these cycles represents logic in one subsystem. It helps to orientate in the pile of names of blocks.

7. Conclusion

The aim of this thesis was to analyse the data flow in the fly-by-wire system. The fly-by-wire system contains a software composed of many models. Because a model is a specific block diagram, it was decided to use graph theory to enable the analysis.

Description of relevant part of the graph theory forms an important part of this thesis. Terms of graph theory necessary for description of the problem of the analysis and for description of the suggested solution are defined in chapter 3. The problem of the analysis of the data flow in the fly-by-wire system was identified as a need of being able to find a path between two blocks within the system of models. In the graph theory search path algorithms are known, and some of them are described in the chapter 4.

Honeywell fly-by-wire system is implemented as a set of Simulink models. Intuitively one would consider a model, or in other words a block diagram, being a graph with blocks being its vertices and lines between blocks being its edges. However, due to possibility of setting different parameters to Simulink blocks, it is not possible to identify a model as a graph. For reasons such as factual link being where none line is in a model, or a model being layered, or one line representing several lines, it is necessary to construct a graph according to the model. The graph has to contain all the information as the model does, and it has to solve the mentioned problems why a model cannot be used as a graph. The analysis of Simulink models and how a graph is creating according to the models is described in the chapter 5.

In cooperation with Honeywell engineers the problem of searching paths within the set of models was refined to a need to decide whether the found path is primary or secondary. In a primary path the data of the start block of the path affect the data of the end block directly, while in a secondary path the end block data are affected indirectly. To be able to distinguish the primary and secondary path within use of algorithms, the algorithm had to be modified. The modification lies in redefining the lenght of path. Instead of additive approach common in the graph theory, the maximal approach is used. This is described in the chapter 6.2.

The rules for creating a graph according to models and the modified algorithms were implemented and tested. As it shows up, the algorithms BFS and DFS are suitable for solving the problems, while algorithm of Dijkstra and algorithm of Floyd and Warshall are not. The discussion of results of the testing is presented in chapter 6.3. For the programs a brief manual is described in chapter 6.4.

As it emerges, the use of the graph theory for analysing the data flow in the fly-by-wire systems is suitable as it was guessed in the beginning of the effort.

Bibliography

[1] Barr, M., Massa, A., *Programming Embedded Systems with C and GNU Development Tools*, CA, USA, O'Reilly Media, 2007, 2nd edition, ISBN 0-596-00983-6.

[2] BRIERE, D., TRAVERSE, P., *AIRBUS A320/A330/A340 Electrical Flight Controls*, Toulouse, France, 1993.

[3] CORMEN, T. H., et al., *Introduction to Algorithms*, The MIT Press / McGraw-Hill, 2001, 2nd edition, ISBN 0-262-03293-7 (MIT Press), ISBN 0-07-013151-1 (McGraw-Hill).

[4] DEMEL, J., *Grafy*, Praha, SNTL, 1988.

[5] HEATH, S., *Embedded Systems Design*, Newnes, 2002, 2nd edition, ISBN 0750655461.

[6] JUNGNICKEL, D., *Graphs, Networks and Algorithms*, Springer Verlag, 3rd edition, 2007, ISBN 3540727795 .

[7] KOLÁŘ, J.; ŠTĚPÁNKOVÁ, O.; CHYTIL, M., *Logika, algebry a grafy*, Praha, SNTL, 1989.

[8] MATLAB [computer program], verze 7.1.0.246 (R14), The MathWorks, 2005.

Websites

[9] ARTIFICIAL STABILITY & FLY-BY-WIRE CONTROL, *ARTIFICIAL STABILITY & FLY-BY-WIRE CONTROL* [ONLINE], Available at http://www.ausairpower.net/AADR-FBW-CCV.html [Accessed 24 May 2010].

[10] Design News, *Model-Based Design Helps Aerospace Engineers* [ONLINE], Available at http://www.designnews.com/article/279138-Model_Based_Design_Helps_Aerospace_Engineers.php [Accessed 24 May 2010].

[11] Developer Zone - National Instruments, *Shortening the Embedded Design Cycle with Model-Based Design* [ONLINE], Available at http://zone.ni.com/devzone/cda/tut/p/id/4074 [Accessed 24 May 2010].

[12] Graphviz, *Graphviz* [ONLINE], Available at http://www.graphviz.org/ [Accessed 24 May 2010].

[13] MathWorks - MATLAB and Simulink for Technical Computing, *Matlab - The Language Of Technical Computing* [ONLINE], Available at http://www.mathworks.com/products/matlab/ [Accessed 24 May 2010].

[14] www.perl.org, *The Perl Programming Language* [ONLINE], Available at http://www.perl.org/ [Accessed 24 May 2010].

[15] Wikipedia, the free encyclopedia, *Aircraft flight control system* [ONLINE], Available at http://en.wikipedia.org/wiki/Aircraft_flight_control_system [Accessed 24 May 2010].